Parallel Computing - Work Assignment 2

Implementing Parallelism on a molecular dynamics' simulation code

Novembro 29, 2023

José Fonte
Universidade do Minho
a91775

Bernard Georges
Universidade do Minho
pg53698

Abstract

This assignment focus on implementing parellelism in a simple molecular dynamics' simulation applied to atoms of a given gas.The goal is to enhance the computational efficiency of the simulation by leveraging the capabilities of modern multi-core processors. This report meticulously outlines the process of incorporating OpenMP into our initial sequential code, resulting in a remarkable reduction in execution time from 37 seconds to 2.05 seconds.

1 | Introduction

In the ongoing evolution of our project performance, our team has undertaken the task of enhancing the efficiency of our existing codebase. To achieve this, we have chosen to leverage the power of OpenMP, a multiprocessing programming interface that facilitates the integration of threads into the code designed sequently in the previous stage. This report serves as a comprehensive documentation of the decisions made by our team throughout the parallelization process.

As a baseline of performance we have the code developed in the previous phase of the project, which follows a sequencial execution. As seen in Figure 3, the execution time is around 70 seconds, therefore in order to improve it the group followed this workflow:

- identify the application hot-spots (code blocks with high computation time);
- analyse and present the alternatives to explore parallelism within the hot-spots identified;
- select an approach to explore parallelism, justified by a scalability analysis;
- implement and optimise the approach;
- measure and discuss the performance of the solution.

2 | Identifying Hot Spots

The initial phase involves pinpointing the critical areas of the code that necessitate improvement and in order to achive this the group employed the "perf" profiling tool. Following the execution of the sequential version of the code, the profilling results, shown in Figure 2, directed our attention to the crucial bottleneck — the *"joinedPotentialComputeAcc()"* function.

3 | Analising Parallelism Alternatives

As denoted in Listing 1, the sequential code within the hot-spot identified presentes three *for* loops (one of them nested inside another) that can be parallelized. The first one is used only to initialize a matrix of Nx3. The second and third(nested) loops are where the entirety of the calculations happen, this implies that this block is where there are the most computations.

There a many ways to deploy for loop parallelization as the **Loop Schedueling** alters greatly the efficiency of the operation. The options made avaliable by the OpenMP are *static*, *dynamic*, *guided*, *runtime* and *auto*. Each type has its advantages and disadvantages which the group had to consider in order to best fit the use case.

Although the scheduling is important the most crutial part of this project is the prevention of data races. This error is caused by multiple threads accessing the same variable in an inordarly way compromising its values in the shared memory. Due to the thread not being correctly synchonized this may lead to unpredictable and erroneous behavior. To prevent this the group needs to verify one of two options:

- Wrapping Variables in a Private Scope: If the variable is only used inside the thread and its values are unimportant to the rest of the code, each thread can have its own registry of these values. This can be assured through the use of the *private* key word or the initialization of such values inside the thread its self.
- Supervision of the values: In case the variable is global or needed by multiple threads there is the option to signal the zone as critical, this means that there can only be one thread at a time executing the signaled code. This option can be done in OpenMP using the *atomic* or *critical* keyword. Although this practice corrects the data race it creates a bottle-neck as threads needs to wait to enter the critical zone.

  An alternative to this method is use the *reduce(func:var)* funtion that creates a private variable for each thread an as the threads the variables are "joined" using the function indicated.

4 | Our Approachs & Implementation

4.1 | First *for* Loop

Due to its low computational weight the group speculated that the use of threads on this block would be unproductive as the synchronisation overhead would surpass its gains.

4.1.1 | Scalability Analysis

Analysing the code, the group observed that in each iteration variables are independent and the workload uniform, this being said, implementing the first approach (parallelizing) would logically lead to a static schedule, which benefits uniform workloads. This approach is intrinsically associated

to an execution overhead that scales as the units involved scale. After running some tests, the group observed that without parallelization the performance has a slight increase as proven by Figure 4.

4.1.2 | Implementation

In the first *for* loop, the group choose to keep the same sequencial code, given that based on our scalability analysis we concluded that the overhead created outweights the benefits of parallelization.

4.2 | Second & Third *for* Loop

Upon initial inspection, the nested loop exhibits imbalances, with each iteration successively reducing in size, consequently, later threads bear significantly lighter workloads compared to their earlier counterparts. Static scheduling is therefore discouraged, prompting a recommendation for a thorough analysis of dynamic and guided scheduling methods.

The group also observed that all variables within the loop are susceptible to data races, with particular attention drawn to the "PE" and "a" variables, given their global scope.

Finally our group decide to parallelize the outter loop to reduce the synchronisation overhead that would happend at every iteration of the outter loop. This also increases the memory wall as parallelizing the outer loop allows for better cache utilization. Each thread can load a portion of the a and r arrays into cache and work on that portion for many iterations of the inner loop, reducing cache misses.

4.2.1 | Scalability Analysis

Dynamic and guided scheduling both involve dividing iterations into "chunks" assigned to threads upon completing previous chunks. The key difference is that guided scheduling reduces chunk size as execution progresses, promoting a more even load balance, especially in unbalanced loops. However, this reduction in chunk size increases the computational requirements for assigning threads. In our tests, dynamic scheduling showed a slight one-second improvement over the guided protocol, prompting our choice to be the implementation of dynamic scheduling.

4.2.2 | Implementation

Based on the Scalability Analysis, the group implemented the following techniques to address each problem. In the second *for* loop, dynamic scheduling was adopted along with the Reduction of incremental/decremental variables. In the third *for* loop, variables were declared within the private scope of each thread combined with a slight code restructure - to decrease cache reads and improve legibility.

5 | Results Analysis

In order to test the solution developed, the group tested executing with an increased number of cores in the cluster, where each core represents one thread.

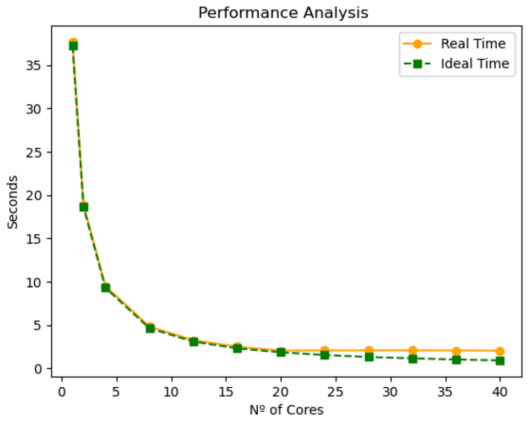

Figure 1: Plotting the Real vs Ideal Time

The Ideal Time is derived from dividing the sequential code execution time by the number of cores, serving as a theoretical benchmark. Meanwhile, the Real Time represents the results obtained from tests conducted by our team on the SeArch Cluster.

As depicted in the Figure 1, real-time performance aligns closely with ideal time, showing expected variance, until reaching a stage of uniform performance. This occurrence is elucidated by Amdahl's Law and the impact of parallelization overhead.

Amdahl's Law asserts that the speedup of a program is constrained by the portion of the code resistant to parallelization, which means the inherent sequential code imposes a performance ceiling. Additionally, the increase in core count can introduce overhead in terms of communication and coordination between threads. This overhead at some point counterbalances the gains achieved through parallelization.

6 | Conclusion

In conclusion, the team's approach to parallelization stands as a robust method, effectively optimizing the performance of our sequential code, from 37 secs to 2.05 secs . The success of our optimization endeavors highlights the efficacy of OpenMP in enabling efficient parallel execution. This project not only serves as a testament to the impactful implementation of targeted parallelization but also underscores the nuanced balance required between sequential and parallel execution for achieving optimal performance.

7 | Attachments:

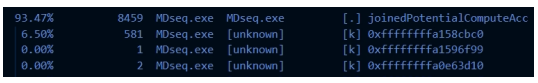

Figure 2: Perf Profiling of the Sequential Code

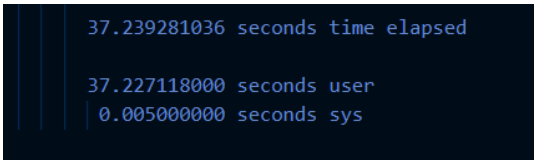

Figure 3: Execution time of the Sequential Code

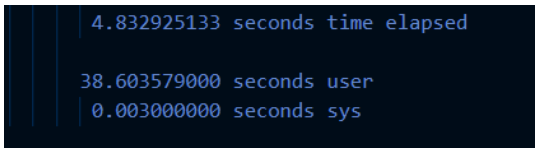

Figure 4: Execution time with first *for* loop parallelized

```
void joinedPotentialComputeAcc(){
    for(i=0; i<N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    PE = 0.;

    for (i = 0; i < N-1; i++) {
        for (j = i+1; j < N; j++) {
            rij[0] = r[i][0] - r[j][0];
            rij[1] = r[i][1] - r[j][1];
            rij[2] = r[i][2] - r[j][2];

          rSqd = sigma6 / (rij[0] * rij[0] + rij[1]
* rij[1] + rij[2] * rij[2]);

            term2 = rSqd * rSqd * rSqd ;
            term1 = term2 * term2;
            PE +=8*epsilon* (term1-term2);

            f = term2 * rSqd * (48 * term2 - 24);

            a[i][0] += rij[0] * f;
            a[j][0] -= rij[0] * f;
            a[i][1] += rij[1] * f;
            a[j][1] -= rij[1] * f;
            a[i][2] += rij[2] * f;
            a[j][2] -= rij[2] * f;
        }
    }
}
```

Listing 1: Sequential Code from *joinedPotentialComputeAcc()*

```
void joinedPotentialComputeAcc(){
    int i;
    double sigma6 = sigma*sigma*sigma
    *sigma*sigma*sigma;

    for(i=0; i<N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }

    #pragma omp parallel for schedule(dynamic, 40)
reduction(+:PE) reduction(+:a[:N][:3])
    for (i = 0; i < N-1; i++) {
        PE = 0.;
        for (int j = i+1; j < N; j++) {
           double f, f0, f1, f2, rSqd, term1, term2;
            double rij[3];
            rij[0] = r[i][0] - r[j][0];
            rij[1] = r[i][1] - r[j][1];
            rij[2] = r[i][2] - r[j][2];

           rSqd = sigma6 / (rij[0] * rij[0] + rij[1]
* rij[1] + rij[2] * rij[2]);
            term2 = rSqd * rSqd * rSqd ;
            term1 = term2 * term2;

            PE +=8*epsilon* (term1-term2);

            f = term2 * rSqd * (48 * term2 - 24);
            f0 = rij[0] * f;
            f1 = rij[1] * f;
            f2 = rij[2] * f;

            //  from F = ma, where m = 1 in natural
units!
            a[i][0] += f0;
            a[i][1] += f1;
            a[i][2] += f2;
            a[j][0] -= f0;
            a[j][1] -= f1;
            a[j][2] -= f2;
        }
    }
}
```

Listing 2: Parallel Code from *joinedPotentialComputeAcc()*